Security in context: analysis and refinement of software architectures

Thomas Heyman, Riccardo Scandariato and Wouter Joosen
IBBT-DistriNet, Katholieke Universiteit Leuven
3001 Leuven, Belgium
thomas.heyman@cs.kuleuven.be

***Abstract*—Security analysis methods can provide correct yet meaningless results if the assumptions underlying the model do not conform to reality. We present an approach to analyze the security of software-intensive system architectures that focusses on making these underlying assumptions explicit, so that they can be taken into account. Starting from an Alloy model of a software architecture, a set of constraints is elicited by leveraging model relaxation techniques. These constraints form a minimal but sufficient condition that the system must meet in order to realise its security requirements. As the approach starts from the minimal guarantees that the system environment offers, it does not depend on an explicit attacker model and can take arbitrary attacker behaviour into account. As it is iterative, it is possible to constructively integrate the approach in a secure software development life cycle. Our results are illustrated by means of a case study.**

***Keywords*-security; software architecture; analysis; Alloy**

I. Introduction

Creating a software architecture is a key step in developing a successful software system, as it is essential in ensuring that the resulting system realizes certain qualities, such as performance, adaptability and security [1]. Also, early studies have shown that it is five to one hundred times cheaper to fix fundamental flaws in the system early on [2], making architectural analysis more cost-effective than finding and fixing bugs after the software is deployed. Given the rising importance of software security, analyzing the security properties of a system on the architectural level is therefore paramount.

Methods to analyze the security of a software architecture can be roughly grouped in two categories: formal methods (e.g., [3]) and threat modelling (e.g., [4]). When an analysis method states that an architecture is secure, the result is always relative to the set of assumptions used. Clearly, if these assumptions are not realistic, the result is formally correct but practically meaningless. In threat modeling approaches, assumptions are implicitly made by the analysts while assessing the relevance of a potential threat to a system. For instance, the threat that information is disclosed over a connection is discarded because that communication channel is behind a firewall. In formal methods, the assumptions are embedded in the model used for the analysis and are often implicit. For instance, availability weaknesses are not discovered if the model can not represent the case of failing components. In both cases, there is no explicit enumeration of the complete set of assumptions that are necessary to back up the analysis result and to build a supporting assurance argument. This work focusses primarily on this concern, which is neglected in the state of the art.

This paper presents a formal architectural modelling and analysis method that iteratively constructs the set of assumptions (called the context) which is required by the system to operate correctly. We provide an analysis method that formally verifies that the system model upholds the security requirements in light of the stated assumptions. The security analyst can use the context as an explicit target of assessment in a risk management process. The context refines the important, yet vague question "Is my system secure?", to a more practical one, "Is the expected context matched by reality?". Additionally, the application deployer can use the context as a checklist of properties that the deployment environment should uphold for the application to be deployed securely.

Our approach is illustrated in Figure 1. First, from the architectural documentation (which is supposed to contain at least information on the system decomposition, the deployment description and the functional scenarios), a formal model in Alloy is derived (see the top part of the figure). The model is created in accordance to the meta-model, which provides architectural abstractions such as components, connectors, and so on. The security requirements are also formally modelled as Alloy assertions. Then, via a model finding technique, a context is iteratively built which contains a formalization of all the assumptions that are necessary for the requirements to be realized. The context is constructed by iterating over all automatically generated counterexamples that violate the requirements of the modelled architecture, debunking the implicit assumptions that made these scenarios feasible, and explicitly adding constraints to the context to prevent these counterexample from occurring (see arrow 1). These constraints represent the expectations on the deployment environment (e.g., the security guarantees provided by the middleware), on the 3rd party security solutions integrated in the architecture (e.g., an authorization engine), and on the internal parts of the system that are not explicitly modelled at the current level of refinement.

Every constraint in the context documents something that the architect explicitly takes for granted in the current formal

model. It is then up to the security analyst to accept these assumptions, or to ask the architect to extend or refine the architecture with additional mechanisms so that the assumption becomes superfluous (see arrow 2). Note that, as the abstractions in the formal model (which are also used in the counterexamples) are very close to the abstractions used in the architectural model, it is easy to interpret the counterexamples and to map the results of the analysis back to the actual issues in the architectural model. Therefore, the proposed approach provides constructive support to the architect as well.

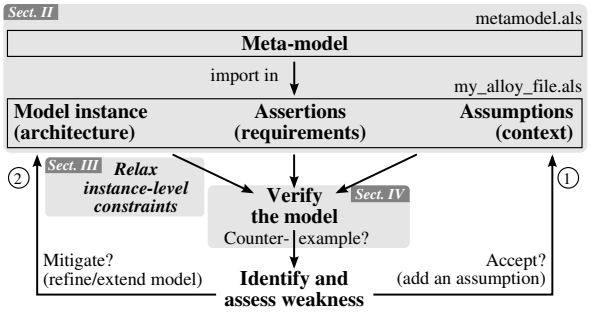


Figure 1. An overview of the approach. The black rectangles denote input files, the arrows denote the steps in our methodology. Sections describing a particular step are marked.

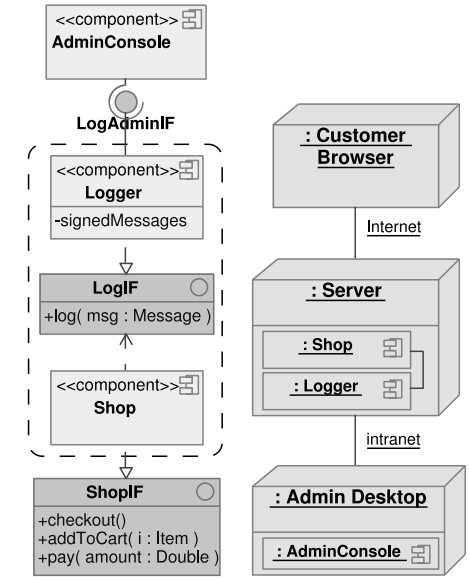
This paper is structured as follows. Section II describes our architectural meta-model, introduces Alloy as the target formalism for our analysis, and shows how architectural descriptions are mapped to Alloy. Section III introduces model relaxation and its impact on verifying the system security requirements. The analysis process and the iterative construction of the context are outlined in Section IV. The class of weaknesses that can be discovered are discussed in Section V and compared to related work in Section VI. We conclude in Section VII.

II. A SECURITY-ORIENTED MODEL OF SOFTWARE ARCHITECTURES

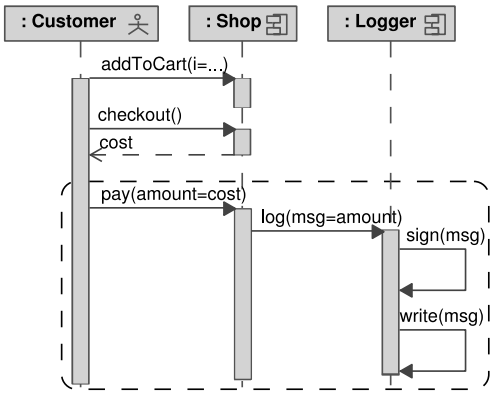
This section introduces the security-centric software architectural model used during the remainder of this paper. In order to illustrate the approach, we apply it to a case in which an online shop needs to securely log transactions, i.e., logged transactions should not be modified nor deleted without being detected. For brevity, only the key parts of the case are presented.

A. Important architectural views

Software architectures are documented in terms of different views, with each view consisting of one or more models [5]. For the purpose of security analysis, we propose a meta-model (based on the "4+1" views from Kruchten [6]) that incorporates information from the physical, logical and scenario view. Information from the development view is disregarded, as we focus on the security of a system running in a specific deployment environment and less on how the system was constructed. While our approach does not require precisely these views (based on available architectural documentation, less or more information could be included in the analysis), we find that they are a practical choice, given the wide acceptance of the "4+1" model on which they are based. We return to this topic in Section V. The viewtypes are applied to the online shop case study in Figure 2 using UML2 notation.



(a) Logical view. (b) Physical view.



(c) Scenario view.

Figure 2. Different views on the online shop.

The *logical view* captures the component model of the system. As shown in Figure 2(a), the logical view of the online shop consists of a Shop component, which realizes the functionality expected by the shop's customers, and a Logger component, which realizes the security requirements imposed on the transaction log. Such a solution could have been selected from a catalog of security mechanisms, such as the Secure Logger security pattern from [7]. The

Logger exposes two interfaces, one public LogIF which is used by components that need to log messages, and a private LogAdminIF which is used by the AdminConsole. The AdminConsole component represents the dashboard that the shop administrator uses to read the log, and to verify that it is still intact. The main purpose of including information from the logical view in the model is to enable the functional modelling of the system (i.e., to create a model that simulates the behaviour of the final system, up to a certain level of abstraction). As such, the logical view has a supporting function, and does not help in uncovering security weaknesses directly.

The *scenario view* illustrates how the elements in the other views work together by the use of a small set of important scenarios, which are instances of more general use cases. Figure 2(c) captures the scenario of buying an item from the shop. The customer adds items to the shopping cart, after which the cart is checked out. The Shop calculates the total cost, and returns this to the customer. The customer pays the specified amount, after which the transaction is logged by the Shop via the Logger. This view builds upon the foundations in the logical view to allow reasoning about what the system should do. It shows which model elements are relevant for the execution of a system operation and enable tracing the impact of low-level security properties to a global impact on the functionality of the system. For instance, the impact of the inability of the Logger to log messages is made clear in Figure 2(c), which shows that the log operation is used in the implementation of the pay-operation of the online shop. In general, the scenario view allows reasoning about architectural security properties such as full mediation, ordering of operations and timing issues such as race conditions.

The *physical view* describes the mapping of the software onto the hardware and reflects its distributed aspects. In Figure 2(b), three nodes are relevant to the shopping system. The customer's browser machine (hosting the browser process which is not depicted) is attached to the server (hosting both the Shop and the Logger process) via the Internet. The Shop and Logger are connected via a local connector, which we model as a link, since UML2 lacks explicit connector semantics. The AdminConsole executes on the administrator's desktop, and is connected to the server via a private intranet. The physical view is the main driver of the security analysis process. The information contained therein allows reasoning about security issues such as data and communication integrity, data flow properties, availability of nodes and links, and the introduction of malicious entities in the deployment environment. For instance, Figure 2(b) allows the analysis to reason about data flow between the customer's browser and the Shop, as it shows where these processes are hosted and which communication channels are available to exchange this data.

B. Alloy

Alloy is a modelling language based on first-order logic [8]. The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking by finding model instances that correspond to a certain model description. Furthermore, the analyzer is able to automatically verify model properties and visualise counterexamples. In order to make the analysis tractable, however, the modeller has to provide an explicit scope, i.e., the number of instances of types defined in that model that are used in the analysis. Only counterexamples within that scope will be found. We use Alloy as it is designed explicitly with object-oriented analysis in mind, facilitating the adoption by the software architect.

As opposed to model checking, where all reachable states of a complete model are visited, model finding takes a set of constraints and finds all models within the scope that uphold these constraints. This is key to taking unforeseen attacks into account, and removing the need for an explicit attacker model. How this is leveraged is shown in Section IV. Also, as Alloy supports analyzing incomplete models, a more flexible integration of our approach in the development process is possible (i.e., the architect can perform an initial analysis on a partial architectural model).

C. Integrating architectural views in an Alloy meta-model

We facilitate the creation of architectural models in Alloy by providing an architectural meta-model, depicted in Figure 3, that incorporates the three architectural views described in Section II-A.

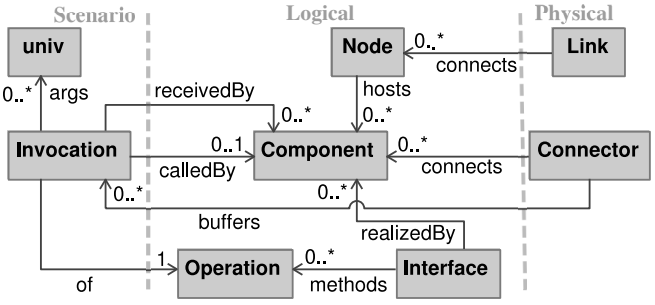


Figure 3. An architectural meta-model in Alloy.

The concepts presented in the meta-model represent a proper subset of UML that we expect the architectural description to adhere to. A software architecture as described in Section II-A is mapped to our Alloy meta-model as follows. The mapping of logical and physical elements is straight-forward. An architectural component, interface and operation are mapped to their namesake types in Alloy, as are UML nodes and links. Architectural connectors are a bit more problematic, as there is no first-class connector in UML. We assume that a connector must be present between two (or more) components, i.e., connect those components,

before they can interact. Furthermore, if a connector exists between components on different nodes, then there must be a corresponding link between those nodes to host the connector. These assumptions are sufficiently generic that they are applicable to every component-based system.

A complete presentation of our Alloy meta-model is not possible due to space limitations[1]. As an illustration, the modelling process is applied to the parts of Figure 2 marked by a dashed rectangle.

Concerning Figure 2(a), the Logger and Shop components are modelled as follows. Note that Connector, Link and Node are defined in the meta-model file (metamodel.als) and omitted for brevity.

```
1 sig Logger extends Component {} // A 'sig' is a type in Alloy.
2 // Alloy supports inheritance, denoted by the 'extends' keyword.
3 // For instance, in this fragment, both Shop and Logger are
4 // specializations of the Component type (not defined here).
5 sig Shop extends Component {
6   logger: Logger one →Time // 'logger' is an attribute of Shop.
7   // It contains a reference to a logger−instance.
8 }
9 sig Log extends Operation {} // Log is an operation
10 sig LogIF extends Interface {}{ // LogIF is an interface.
11   // The LogIF is realized by the Logger, and contains
12   // only one method (i.e., 'Log'). The methods and realizedBy
13   // relationships are defined in (and inherited from) Interface.
14   methods = Log and realizedBy = Logger
15 }
```

Before the Shop and Logger components can be detailed, we need to define the types that are used in their definition. As shown in the LogIF (line 10), the Logger accepts Messages via the Log operation. Internal to the Logger component, a logged message is represented in a different format (e.g., a sequence number is added). These additional details can be derived from the description of the adopted solution (e.g., the Secure Logger pattern [7]). To this aim, the ProcessedMessage type is introduced (line 16). According to a similar rationale, the SignedMessage type (which denotes a digitally signed ProcessedMessage) must be introduced as well (line 22). It contains a reference to the current ProcessedMessage (line 23), the signedContent (line 24), i.e., the ProcessedMessage that was originally signed, as well as the Principal that signed the message (line 25). A SignedMessage is valid at a certain point in time when the content matches the signed content, and it was signed by the LoggerEntity. This is captured in the isValid predicate on line 30. Note that the above Alloy description of the Secure Logger pattern could be provided to the modeller via a library.

```
16 sig ProcessedMessage {
17   // 'content' is a relation from one Message to Time atoms.
18   content: Message one →Time,
19   // Similarly, 'id' is a relation from one integer to time.
20   id: Int one →Time
21 }
22 sig SignedMessage {
23   content: ProcessedMessage one →Time,
24   signedContent: ProcessedMessage one →Time,
25   signedBy: Principal one →Time
26 }
27 // This defines a predicate on instances of SignedMessage,
28 // with one Time 't' as argument. Note that, as in Java,
29 // the current instance is accessible with the keyword 'this'.
30 pred SignedMessage.isValid(t:Time) {
31   // In Alloy, relationships can be navigated using '.'.
32   this.content.t.content.t = this.signedContent.t.content.t
33   this.signedBy.t = LoggerEntity
34 }
```

Now that the necessary types have been introduced, the previous definition of the Logger can be completed with the attribute shown in Figure 2 (i.e., a mutable list of signed messages) as follows.

```
35 sig Logger extends Component {
36   // 'contains' attributes a variable list of SignedMessage
37   // instances to every Logger instance.
38   contains: set SignedMessage →Time
39 }
```

As Alloy is based on first-order logic, it does not have a built-in notion of time. As such, time is explicitly modelled as a set of discrete, ordered Time instances. Associations (such as the set of nodes connected by one link) can be made mutable by adding a relationship with the Time set (i.e., the 'connects' relationship between a Node and a Link in Figure 3 is in $\{Node \times Link \times Time\}$).

Central to the scenario view is the Invocation, which can be created by the calling component (i.e., invoked) and consumed by the receiving component (i.e., executed). Once an invocation is invoked, it is buffered in a connector. When it is consumed by the receiving component, it is removed from the connector. The details of these semantics are are included in the meta-model (metamodel.als) and not shown here.

```
40 sig Invocation {
41   of: one Operation,
42   args: set univ,
43   // The caller, as expressed 'on the wire'.
44   caller: lone Component,
45   // The component that sent the original invocation.
46   orig_caller: one Component,
47   receivers: set Component, // The receiving components.
48   invoked: lone Time, // The time of invocation (if any).
49   executed: lone Time // The time of execution (if any).
50 }
```

An equivalence relationship is defined between invocations with identical senders and receivers, the same operation, and the same invocation time. This allows analyzing situations where an invocation is swapped for an equivalent one, e.g., one with the same logical caller (line 46) but actually sent by a different physical caller (line 44), as is the case

[1]The complete, extensively documented Alloy source files can be found at http://people.cs.kuleuven.be/thomas.heyman/alloy/.

when an invocation is routed through different intermediary components, for instance.

Every operation is coupled to its behavioural specification by adding an extra fact to the model. In architectural terms, this corresponds to associating a request received at a component interface to the internal implementation of such an operation. For instance, consider the Logger which offers a Log operation. Suppose that the specification of the Log operation is contained in the predicate Logger.write, then the Log operation is coupled to this specification as follows (note line 65).

```
51 pred Logger.write(m:Message, t:Time) {
52   // 'some' represents existential quantification in Alloy.
53   some pm:ProcessedMessage, s:SignedMessage {
54     pm.content.t = m and s.content.t = pm
55     s.sign[LoggerEntity,t] and s in this.contains.t
56 }}
57 pred Logger.log(callr:Component, m:Message, t:Time) {
58   // The Call predicate creates a corresponding Invocation.
59   Invoke[callr,this,Log,m,t] }
60 sig Logger extends Component { ... }{
61   // 'all' represents universal quantification.
62   all t:Time | some c:Component,arg:univ |
63     // The Execute predicate is true whenever a
64     // corresponding Invocation is executed.
65     Execute[c,this,Log,arg,t] ⇒ this.write[arg,t]
66 }
```

The Shop is now able to invoke the Log operation by calling `Logger.log(this,m,t)`, with `m` the message contents, and `t` the current time instance. Note that, since Alloy does not offer reflection, it is not possible to couple the behaviour of an operation directly to the instance of an Operation model element.

The architect can instruct the Alloy Analyzer to generate instances of the model with the command `run {} for 3`. This will visualise arbitrary model instances within a scope of 3 (i.e., with a maximum of three component instances, three connector instances, three times, etc.). Larger scopes allow reasoning over more complex instances, while increasing analysis overhead. By checking that the model still has instances within the current context, the architect can verify that the model is *realizable* (i.e., that there are no internal contradictions). This check enforces the consistency of our approach.

D. Security requirements

Supposing that the rest of the system of Figure 2 is similarly modelled, it is now possible to express and verify the security requirements of the Logger. We assume that the security requirements are expressed in terms of the interface exposed by the respective component (as in [3]). The integrity requirement that messages written to the log should not be deleted, is modelled as an assertion.

```
67 assert NoDeletion {
68  all t:Time,m:Message,s:Shop | (s.logger.t).log[s,m,t] implies (
69   some pm:ProcessedMessage,t1:t.nexts + t {
70     pm.content.t1 = m and all t2:Time | t2.gte[t1] implies
71       (s.logger.t2).checkForEntry[pm,t2] or
72       (s.logger.t2).isIntact[0,t2]
73 })}
```

Note that the checkForEntry and isIntact predicates represent the implementation of the Read, resp. Verify operations. The predicate checkForEntry is true if the specified processed message is present in the log at that time, and isIntact is true whenever the boolean flag (0 for false, 1 for true) matches the perceived status of the log. In other words, the NoDeletion requirement asserts that an invocation of the Log operation from a Shop results in a correct state of the Logger.

The architect can verify whether the NoDeletion requirement holds with the command `check NoDeletion for n`, which would exhaustively explore every model instance within a scope of n. If it does not hold, a counterexample can be visualised. This counterexample represents an attack against the model. The absence of counterexamples guarantees that the requirements follow logically from the model, within the specified scope. This check enforces that the model is complete w.r.t. the current level of refinement.

E. Meta-model revisited

As the meta-model describes a very generic architectural model, it is easily extended to conform to specific application deployment environments. But, more importantly, this genericness ensures that all weaknesses that are present in a more specific model (with more constraints), also occur in this general one. In the default model, the following properties are assumed **not** to hold. *Liveness* and *at-most-once invocation semantics*. It might be that one invocation gives rise to a repeated execution of the corresponding operation, or no execution at all. *In-order message delivery* or other timing-related properties are also not taken for granted. Furthermore, all modelled entities are not considered *static*—components can dynamically be added to or removed from a connector, communication links can fail and new ones can be introduced, etc.

Most importantly, models are not assumed to be completely described. During analysis, arbitrary unnamed entities such as components and nodes will be introduced. As the only thing that is taken for granted are the constraints expressed by the architect, the analysis on such a generic model will uncover every configuration of model entities that violate the specified requirements, within the constraints imposed by the modeller. This effectively factors out an explicit attacker model, and shifts the focus to a realistic modelling of the deployment environment. This is the topic of the next section.

III. Uncovering hidden assumptions through model relaxation

The modeller may implicitly introduce constraints that are too strong for the deployment environment and that might

fail at run-time. In these cases, a positive result from the security analysis is not reliable as the architecture is "secure" only vis-a-vis a set of unrealistic assumptions. Hence, the analysis tool might even create a counter-productive, ill-grounded belief of security.

Consider, for instance, the following alternative definition of a SignedMessage in which the reference to time-variability has been dropped.

```
74 sig SignedMessage { ...
75   content: one ProcessedMessage // was ''→ Time''
76 }
```

This definition silently assumes that the content of a signed message is immutable, i.e., it is impossible to alter a signed message, as it is not time-dependent. This hidden integrity assumption prevents reasoning about situations where the content of a SignedMessage is changed over time by an attacker, thereby hiding potential architectural weaknesses.

As a more subtle example, consider the following definition of an invocation.

```
77 sig Invocation { ...
78   executed: one Time // was ''lone''
79 }
```

It implies that every invocation is executed at exactly one time. In other words, this model implicitly assumes that all invocations will eventually be executed (so deadlock is not possible), the connector containing that invocation can not fail (e.g., because of denial of service), there is always a time when the receiving component is available (and, a fortiori, there is a node hosting that component) and, if the sender is a component hosted on another node, the link between both nodes is available.

The above examples clarify how easy it is to introduce implicit assumptions in an architectural design and that the security implications can be far-reaching. Generally, hidden assumptions can be avoided on two levels. The first level is procedural, and consists of enforcing certain relaxation rules, described later. The second level is structural, and consists of embedding the relaxation rules in the meta-model (to a large extent). This way, whenever a model is created in conformance to the meta-model, many hidden assumptions can be avoided by construction. This is an advantage for the modeller, as enforcing the relaxation rules can be challenging, especially for large-scale systems.

A. Relaxation rules

While it is hard to provide completeness criteria for these relaxation rules, practise has shown that the following rules avoid introducing hidden assumptions in new models. *Avoid over-constraining the cardinality of relationships.* Both connectors and links should connect an arbitrary (i.e., $0..*$) number of components, resp. nodes, by default. This permits reasoning about confidentiality and availability issues, such situations where a connector fails and does not connect the necessary components, or where an unforeseen component is accidentally connected. Second, *avoid over-constraining the number of instances of components and nodes*. It should be possible to have too few (e.g., no logger is present in the system), as well as too many instances (e.g., there are two loggers) of every type. This relates to availability and, to a lesser degree, integrity issues. Third, *do not assume that relationships are constant throughout time*. This prevents reasoning about situations where component attributes are altered, or entities are spoofed. This rule mainly relates to integrity issues. Fourth, *avoid making timing assumptions*. This includes liveness assumptions (i.e., assuming that something will happen in every model instance) and atomicity assumptions (i.e., assuming that two events happen atomically). This rule mainly relates to availability issues, and in a lesser degree to integrity issues (race conditions etc.). Last, *avoid assuming that events are ordered*, such as assuming that invocations are delivered in order. This mainly relates to integrity issues, including race conditions.

B. Supporting relaxation via the meta-model

Correctly applying these relaxation guidelines might be cumbersome for the modeller. Fortunately, most of the model relaxation rules are embedded at the meta-model level, so that they are fulfilled by construction, making the model relaxation process as transparent as possible. This is done as follows. First, the relationships between different meta-model elements are not constrained in cardinality, e.g., a connector can connect an arbitrary amount of components. Additionally, wherever applicable, cardinalities are generalized to 'lone' ($0..1$) as opposed to 'one'. This is the case for Invocations, that might be invoked (at a lone Time) and might be executed (at a lone Time). Second, the amount of instances one type can have is not constrained in the meta-model. Third, all relevant relationships defined in the meta-model are time-dependent, except for the attributes of Invocation. However, the equivalence relationship defined between Invocations fulfills a similar role, as an invocation with altered arguments will be considered equivalent to the original one. Fourth, the only timing assumption built-in to the meta-model is that an Invocation can only be executed after it is invoked. However, not all Invocations are necessarily invoked, and an invoked Invocation is not necessarily executed. Fifth, Connectors do not guarantee in-order invocation delivery (or even delivery at all, for that matter). All this ensures that "vanilla" instantiations of the meta-model are relaxed as per Section III-A.

Once the modeller introduces custom types and additional constraints in a model, some rules can no longer be enforced on the meta-model level only. Ultimately, the modeller is responsible for not overly constraining the model description, e.g., the modeller should take care that newly introduced component attributes are mutable by making them time-dependent. This is where the relaxation rules

should be applied manually. However, it is possible to facilitate enforcing these relaxation rules by means of pattern matching tools (e.g., scan a model description for attribute definitions, and verify that they are time dependent). This is not addressed in this paper.

IV. Constructive model analysis

In order to be usable in practise, the modelling approach needs to be integrated in a constructive process. This is discussed next.

A. Creating an initial model context

In Alloy, asserting whether a property holds corresponds to trying to find examples where the negation of that property holds. The properties of interest to us are the security requirements. Given a requirement R, the Alloy Analyzer will try to find model instances where $\neg R$ is true. If such an instance is found, then that instance is effectively the abstraction of an attack on the model. If no such instance is found, R holds for all model instances, and the architectural model is secure (within the scope of the analysis).

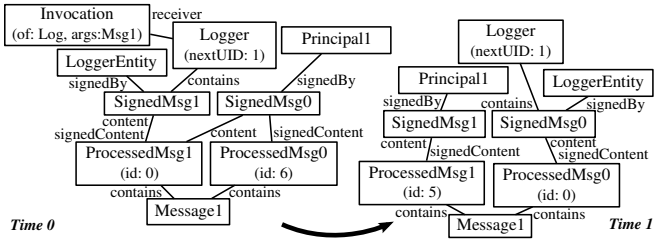


Figure 4. Automatically generated visualisation in which a logged entry is deleted.

When the NoDeletion requirement from the previous Section is verified against our initial model, the tool finds an attack in which a message is logged and simply deleted from the log (i.e., removed from the Logger.contains relationship). In order to mitigate this attack, the explicit assumption that no logged entry can be deleted, needs to be added. This is essentially the negation of the attack instance. As such, the counterexample provided by the tool is an insightful means to constructively extend the architectural model.

```
80 pred DeletionIsImpossible {
81   all l:Logger, t:Time − TO/last, s:SignedMessage |
82     s in l.contains.t ⇒ s in l.contains.(t.next)
83 }
```

When this predicate is added as a fact to the model, deletion attacks are no longer generated. Instead, the Alloy Analyzer finds other counterexamples where the invocation to log a message is never processed by the logger, one where the connector buffering the invocation breaks down before the log-invocation is processed, one where a connector is used by the shop before the logger is connected, one where log calls are routed trough insecure connectors (even though secure connectors exist) and one where invocations are tampered with (i.e., their arguments modified) while transmitted. Analogous to the previous paragraph, facts are added to the model until no additional counterexamples are generated. This iterative process results in the following context, which contains both assumptions on the model instance-level and on the meta-model-level.

```
84 fact Context {
85   DeletionIsImpossible
86   AtLeastOnceInvocationSemantics
87   LogConnectorsAreReliable
88   LogConnectorsCorrectlyInitialized
89   LogCallsCorrectlyRouted
90   LogConnectorsAreTamperProof
91 }
```

Next to the DeletionIsImpossible assumption, this context assumes that the middleware on which the system is built should support at-least-once invocation semantics, so that the logger receives at least one call per invocation of the log-operation. Furthermore, connectors between the shop and log should be reliable (i.e., they do not become unavailable after a set of components is connected), protect the integrity of invocations "in transit", and be correctly initialized (i.e., used only after both the shop and log are connected). Additionally, invocations of the log-operation should be routed through these connectors. This corresponds, e.g., to web applications that, while they support HTTPS, also accept HTTP-calls, rendering surfing sessions of oblivious end users insecure.

Having a sufficiently relaxed model is key for reasoning about what model instances are possible within the minimal guarantees that the modelled deployment environment offers. Every model instance allowed within this space (i.e., configurations of both anticipated and unanticipated components, which exhibit arbitrary behaviour unless explicitly forbidden by some statement in the model description) that violates the requirements is considered as a possible attack.

At this point, it is up to the security expert to decide which parts of the context are trivially true and do not need further attention (i.e., they are accepted) or those that need to be mitigated. Mitigation is done by refining and extending the model with security mechanisms. We argue that it is easier to assess whether a certain deployment environment offers a specific guarantee (positive perspective), than trying to anticipate all possible attacker behaviour in advance (negative perspective). And, when the security expert is unable to easily assess whether a specific guarantee holds in the deployment environment, it is possible to selectively refine parts of the model to facilitate the assessment, as shown next.

B. Refining the model context

Although the initial context presented above might be acceptable in some circumstances (for instance, when the log is written to a "write once read many" medium such as

a CD-R, the assumption DeletionIsImpossible is guaranteed by construction), we will continue from the premise that it is not. In order to refine the context, the part of the model related to the unacceptable assumptions is refined by adding a countermeasure. In this case, the Logger is fitted with a counter, and every ProcessedMessage receives a sequence number between 0 and the value of that counter, which is subsequently incremented. Note that the security requirements to be verified remain unchanged, as they are expressed in terms of the interface of the system, while the mentioned refinement deals with the component internals.

The original course-grained, unacceptable assumption (line 85) is removed from the context, and is refined by means of the guidance provided by the attacks produced by the Alloy Analyzer (lines 92–95).

```
92 fact Context {
93   AssumeTamperproofNextUID
94   LoggerSignaturesCannotBeForged
95   ShopUsesSameLogger
96   LoggerDoesNotOverwriteEntries
97   AtLeastOnceInvocationSemantics
98   LogConnectorsAreReliable
99   LogConnectorsCorrectlyInitialized
100  LogCallsCorrectlyRouted
101 }
```

First, the logger should protect the integrity of the next sequence number to be assigned. Second, signatures of logged entries can not be forged (i.e., if the signed message contents differ from the actual contents, the message signer has to be different as well). This corresponds to assuming that the signature algorithm is secure, which entails that key management happens in a tamper-proof fashion. If the modeller is not sure about the validity of this assumption, it should be refined further. This is not done here due to space constraints. Third, the shop should make sure to use the same logger throughout time. This corresponds to an integrity assumption on a component attribute, as is the case in the first assumption. Fourth, the logger does not overwrite older entries. This corresponds with an assumption of trust in the logger component—the specified behavioural description is complete, and the logger will not exhibit additional functionality. The architectural-specific part of the context remains unchanged.

The *completeness* of the context is trivially verified by the Alloy Analyzer—if no counterexample is found, then the context is complete. In order to verify whether the context (or the model itself) is not overly restrictive, it is important to verify that the model still has instances by effectively generating an example instance. As long as the context is not complete, new assumptions are added one-by-one as new counterexamples emerge.

As the context is built incrementally starting from the counterexamples provided by the tool, it is possible that too many explicit assumptions have been added along the way. In order to verify that the context is *minimal*, the Alloy Analyzer can be used to verify that the removal of any single assumption from the context results in an attack. Note that this only verifies that the context is minimal with respect to the granularity with which the assumptions are expressed—it is possible that a coarse-grained assumption a can be refined in two finer grained assumptions $a = a_1 \vee a_2$ of which only one is necessary. While we have considered normalizing the assumptions to, for instance, a normal form in order to find the absolute minimal set of assumptions within the current level of abstraction of the model, we have chosen not to do this, as it is much harder to interpret normalized assumptions and assess their inherent risk.

By assessing the risk inherent in taking every explicit assumption for granted, the architect receives direct feedback of where security mechanisms are lacking (i.e., the risk inherent to accepting that assumption is too high) or where additional architectural refinement is required (i.e., the risk inherent to accepting that assumption is unclear). This provides the necessary hooks to further integrate our approach in a complete risk management process. However, this aspect of the approach is out of scope for this paper.

The previous discussion assumes that the modeller is creating a context in which the system will operate correctly in a top-down fashion. Conversely, it is also possible to use an existing, pre-packaged model (i.e., a *profile*) which corresponds to the actual deployment environment of the system, and verify whether the architectural model combined with this pre-packaged context results in a secure configuration. For instance, a profile can be developed for the security guarantees provided by a specific middleware (e.g., in-order message delivery, at least once invocation semantics, tamper-proof links). Consider an environment with SSL-protected connectors. This would result in the following predicate provided by the profile (to be added to the context).

```
102 pred SSL {
103   all c:Connector {
104     c.receiversAuthenticated and c.deleteProof
105     c.tamperProof and c.callersAuthenticated
106 }}
```

When deployed in such an environment, our experiments show that to minimize the online shop context lines 97–99 can be safely removed.

V. Discussion

The types of assumptions that can be uncovered (and dually the type of security properties that can be analyzed) depend on the information incorporated in the model and, hence, on the view types described in Section II-A. The current model can unveil *integrity issues*, as all component attributes as well as invocations are mutable; *availability issues*, as the number of element instances is unconstrained; *timing-related issues* such as deadlock, liveness and synchronization issues, as the model implicitly supports parallelism (i.e., multiple invocations can be made per time instance,

and under-constrained parts of the model exhibit arbitrary behaviour). The only caveat is that predicates are treated atomically—if an operation is encoded in a single predicate, its execution will be implicitly atomic.

To verify that non-trivial types of weaknesses can be uncovered, we compare our approach to Microsoft's STRIDE [4], which is a well-known and adopted approach in industry. STRIDE uses a data-flow oriented view on the system (data flow diagrams) and a checklist of generic threat templates, called threat tree patterns. By encoding the converse of each STRIDE threat which is applicable to the meta-model (e.g., spoofing external entities or tampering with data flows) as an assertion and verifying that the Alloy Analyzer finds counterexamples, we illustrate that we achieve at a minimum the same level of analysis. The full set of assertions is included in the online Alloy files. As an example, as data flows are realized via operation invocations in our meta-model, denial of service against a data flow is encoded as follows:

```
107 assert InvocationsCanNotFail {
108   all c1,c2:Component | some p:Principal,op:Operation,
109         t1,t2:Time,if:Interface {
110     op in if.methods and c2 in if.realizedBy
111     Invoke[c1,c2,op,none,p,t1] ⇒ Execute[c1,c2,op,none,p,t2]
112 }}
```

The main advantage of our approach over checklist-based approaches (such as STRIDE) is twofold. First, weaknesses uncovered by this approach are necessarily relevant to the specific security requirements of the modelled architecture. On the contrary, during a checklist-based analysis, the relevance of each generic threat must be assessed in the specific context of the analysed system. Second, the approach abstracts away attacker behaviour by reasoning about the minimal assumptions that are expected to hold in the deployment environment. This is not so in approaches with an explicit attacker model, in which the possible behaviour of an attacker is fixed.

The *usability* of this approach is influenced by four factors. First, because Alloy enables verification of incompletely specified models, even very small initial models can already be used for initial verification. Second, due to the nature of the context refinement process, design effort is focussed on the security-critical parts of the architecture. Fine-grained refinement (possibly down to the level of detailed design) is only required in high-risk situations (when an assumption is unacceptable), justifying the additional effort. This is the case with the next sequence number in the logger component from Section IV-B. Third, because the requirements are expressed in terms of the component interfaces, it is possible to hierarchically validate a complex system bottom-up: once a subsystem is validated and its assumptions accepted, the requirements of that subsystem can be added as assumptions on a higher hierarchical level. However, studying this hierarchical verification is part of ongoing work. Fourth, the effort required to translate architectural documentation to Alloy can be limited using automated model transformations. Given a proper subset of UML diagrams adhering to the meta-model presented in Section II-C, it is possible to largely automate the translation to Alloy. This translation is also part of ongoing work.

The *scalability* of applying this approach is influenced mainly by the run-time verification overhead. Verification performance can be traded off with the exhaustiveness of the analysis via the scope supplied to the Alloy analyzer. Early in the modelling process, smaller scopes can uncover the simpler attacks (the "low-hanging fruit") almost in real-time. Later in the validation process, larger scopes can potentially find very complex attacks, while requiring considerably more verification time. The approach is also made more scalable by limiting verification effort to the security-critical parts of the architecture, and by hierarchically validating subsystems, as mentioned earlier.

VI. Related work

The idea of eliciting and documenting assumptions made during the software engineering process to avoid bugs, is not new. Fickas et al. note that not only requirement fulfilment should be monitored, but also the assumptions under which the system is designed [9]. Wile shows how assumptions can be seen as a form of residual requirements, which remain after an architecture is created to realise a set of original requirements [10]. Roeller et al. give an overview of assumption elicitation in software engineering in [11]. In that work, the authors describe a methodology to recover assumptions made on a technical, organizational and management level, based on analyzing various sources such as interviews, financial reports, version control and source code. Khan et al. observe that software also has to be assessed relative to the context in which it will be deployed [12]. However, as far as we are aware, no methodology exists to (largely) automate the uncovering of context-dependent security assumptions on an architectural level, while providing formal guarantees that the resulting architecture is correct.

Assumptions also exist on the requirements level. Haley et al. claim that adequate security requirements are well defined, explicitly take assumptions into account, and provide satisfaction arguments [13]. We support this by formally encoding requirements as assertions, documenting assumptions in the context, and providing satisfaction arguments by means of model validation. According to KAOS [14], an assumption (or expectation) is a goal assigned to a single agent in the software environment that, unlike requirements, can not be enforced by the software-to-be. This is similar to our approach. However, we have a more narrow scope for assumptions by targetting only the software architecture and provide a constructive method to uncover these assumptions. The results of our analysis, i.e. unmitigated assumptions, can

be mapped to obstacles in KAOS. How we find assumptions is compatible with KAOS, where an obstacle tree is constructed by negating a goal G, and then finding as many AND/OR refinements of $\neg G$ as possible.

The architectural modelling approach most closely related to our approach is the software specification and analysis method (SAM) [3]. SAM focusses on creating a formal architectural representation for simulation, reachability analysis, model checking and interactive proving of architectural properties, but does not help in eliciting assumptions. While our approach supports simulation, model finding and verification of generic properties (within the limits of the Alloy Analyzer), this is not its primary objective.

Using model relaxation techniques on the resulting models is also not new (although rarely used in software engineering). Engler et al. leverage model relaxation and under-constrained execution in software testing [15]. They use these techniques to provide dynamic tools with an ability equivalent to static analysers, by enabling the tools to execute arbitrary functions, without prior setup or environmental modelling. Similarly, our approach does not require an explicit list of initial model states, or to list all possible model transitions. This allows reasoning about situations where the architecture would not be initialised correctly, or where an attacker finds a model transition that the architect did not foresee (e.g., by code injection).

VII. Conclusions

We have presented a light-weight formal approach which leverages model finding techniques to analyze a software architecture and uncover non-trivial types of security weaknesses. The presented approach is constructive in the sense that it provides easy-to-understand counterexamples that make the process of fixing the discovered security issues easier. Furthermore, as a byproduct of the approach, the resulting formal model supports traceability (via the so-called context) of the assumptions made by the modeller on the deployment environment of the system.

Ongoing research extends this work in two dimensions. First, in order to facilitate the integration in a secure development life cycle, tool support for automated translation from UML to Alloy (and back) is being investigated, as well as hierarchical model refinement. Second, we are looking into extending the meta-model to include other architectural view types, such as including a data flow view by extending the model with epistemic constructs. This, in turn, will allow us to analyze more properties. These two extensions will be validated on a larger case study.

Acknowledgment

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[2] B. Boehm and V. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, p. 426, 2005.

[3] X. He, H. Yu, T. Shi, J. Ding, and Y. Deng, "Formally analyzing software architectural specifications using SAM," *The Journal of Systems & Software*, vol. 71, no. 1-2, pp. 11–29, 2004.

[4] M. Howard and S. Lipner, *The Security Development Lifecycle: SDL: A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006.

[5] "Systems and software engineering – recommended practice for architectural description of software-intensive systems," 2007, ISO/IEC 42010:2007.

[6] P. Kruchten, "Architectural blueprints – the "4+ 1" view model of software architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995.

[7] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2005.

[8] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 2, p. 290, 2002.

[9] S. Fickas and M. Feather, "Requirements monitoring in dynamic environments," *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, p. 140, 1995.

[10] D. Wile, "Residual requirements and architectural residues," in *Proceedings of the 5th International Symposium on Requirements Engineering (RE01), Toronto, Canada*, 2001, pp. 194–201.

[11] R. Roeller, P. Lago, and H. van Vliet, "Recovering architectural assumptions," *The Journal of Systems & Software*, vol. 79, no. 4, pp. 552–573, 2006.

[12] K. Khan and J. Han, "Assessing security properties of software components: A software engineer's perspective," in *Proceedings of the 2006 Australian Software Engineering Conference (ASWEC), Apr*, 2006, pp. 18–21.

[13] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security requirements engineering: A framework for representation and analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, 2008.

[14] A. van Lamsweerde, *Requirements Engineering: from system goals to UML models to software specifications*. Wiley, 03 2009.

[15] D. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM New York, NY, USA, 2007, pp. 1–4.